ELSEVIER

The Journal of Logic and Algebraic Programming 51 (2002) 77–100

THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING

www.elsevier.com/locate/jlap

A performance-based methodology to early evaluate the effectiveness of mobile software architectures

Vittorio Cortellessa *, Vincenzo Grassi

Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", 110 via di Tor Vergata, 00133 Roma, Italy

Received 18 May 2001; received in revised form 25 July 2001; accepted 10 September 2001

Abstract

Modern programming languages and hardware technologies require (and effectively enable) ever more software to be distributed. Different paradigms (client–server, mobility-based, etc.) can be adopted to design distributed software, and deciding the "best" paradigm is a typical choice to be made in the very early software design phases. Several factors should drive this choice, that can also change depending on the software application domain. Within this framework, we focus on a class of attributes related to the performance, and the contribution of this paper is twofold: we extend an existing architecture description language (ADL) to model the physical and logical mobility of components; we introduce a methodology that, starting from a software architecture described using this extended notation, generates a performance model (namely a Markov decision process (MDP)) that allows the designer to evaluate the convenience of introducing logical mobility into a software application. The insights gained by applying our methodology do not consist only in deciding whether a mobile paradigm is effective for the performance improvement of a given application, but also how to use the paradigm, that is where a component should move, based on a certain type of performance indices.

Keywords: Software architecture; Architecture description language; Mobile component; Performance analysis; Markov decision process

1. Introduction

One of the first steps in the design of a software application is the definition of its *software architecture* (SA), that is its representation in terms of components and interactions among them [4]. Decisions made at this early stage can have a deep impact on the overall quality of the final software product, since they can affect quality attributes like reusability, performance, and reliability [7,12,27]. For this reason, it is important to develop methodologies that evaluate the impact on these attributes of different architectural choices, at

* Corresponding author.
E-mail addresses: cortelle@info.uniroma2.it (V. Cortellessa), grassi@info.uniroma2.it (V. Grassi).

1567-8326/02/$ - see front matter
PII: S1567-8326(01)00016-9

the time these decisions are made. To this purpose, formal and semi-formal notations have been defined (referred to as "architecture description languages" (ADLs)) for representing SAs [4,16]. These languages aim at providing both a clean syntax for characterizing an SA and tools for analyzing SA quality attributes. In particular, with regard to quality attributes like performance, methodologies have been recently proposed to derive automatically a performance evaluation model from the description of an SA in a given ADL [1,3,8,21,27].

One of the decisions to be taken during the definition of an SA concerns the adoption of an *architectural style*, where a style determines the component types and the patterns of interactions that can be used [4]. Several styles have been identified, motivated by current trends in software design and technologies. In particular, the idea of explicitly taking into account the notion of component *location* in the design of software applications has emerged in recent years. This idea is mainly motivated by the fact that it is becoming increasingly common for applications to operate in large scale computing environments (like the World Wide Web), where locations are both geographically and logically distributed, and interactions among components may have very different characteristics and constraints, depending on the component location. As a consequence, several languages for describing and reasoning about such distributed applications have been proposed [6,9,18,26], where component location is treated as a "first class" element.

From the software architecture viewpoint, the adoption of such a "location-aware" perspective in the design of distributed applications raises the question of which is a suitable architectural style for such applications. As an alternative to the traditional *client–server* style, a style based on the notion of *code mobility* [11] has been suggested, where components of an application may autonomously decide to move themselves to different locations, during the application lifetime. Modern software technologies (e.g. Java) provide tools to implement this style. Arguments in favor of this new style concern the improvements of both qualitative attributes (like customizability) and quantative ones (like network traffic). However, it is also recognized that such arguments are not valid in general, and hence the choice between the two above styles should be performed on a case-by-case basis [11].

Our goal is to define a methodology that helps the designer to perform such a choice at the SA stage, by giving insights into the impact of the two styles (i.e. client–server and code mobility) on the final application. Given the large spectrum of quality attributes affected by such a choice, our methodology does not encompass all of them. Rather, it deals with quantitative attributes that are significant for the class of applications that operate in a geographically distributed environment, with heterogeneous (and possibly physically moving) locations. In particular, we focus on interaction related performance indices (e.g. generated network traffic, possibly on some particular links, or consumed energy, of interest for portable devices [14,22]). For a more comprehensive evaluation of the merits of different styles, our methodology should be extended to cope with different attributes.

As outlined above, methodologies that derive performance evaluation models from the description of an SA have been recently proposed [1,3,8,21,27]. In all these papers the target performance model is a queuing network, and hence the performance indices taken into consideration are those typically evaluated in such a framework, e.g. throughput or response time. These papers consider "static" architectures, where no kind of mobility is present, neither at the level of computing platform (physical mobility) nor at the software component one (logical mobility). Our approach is instead specifically targeted to "mobile" architectures, where mobility can be intended both in physical and logical sense, and deals with static architectures as a special case.

The target model of our performance evaluation methodology is a Markov decision process (MDP) [24]. Such a model embeds features suitable in case of mobility-based applications:

- the MDP probabilistic structure (state transitions and transition probabilities) models the uncertainty about the actual application execution pattern and the execution environment variations (e.g. physical mobility of computing devices);
- the MDP reward structure models the "cost" incurred during the application execution;
- the MDP decisional structure (alternative decisions associated to some states) allows us to model the choice between mobility options, such as "moving" or "not moving", as the implementation of a given *mobility policy*, i.e., a choice between alternative decisions in the process states. We indeed consider the mobility policy as a voluntary choice, aimed at the application working better in a given environment with respect to specific performance indices.

Hence, different from queueing network based approaches, we have chosen a modeling and evaluation framework that allows us to model in a natural way the uncertainty about choosing a code mobility style in the design of software architecture, instead of a static client–server style. Indeed, our target model being an MDP, this uncertainty is modeled as the uncertainty about the policy that optimizes some suitable cost measure.

Papers that evaluate the impact of code mobility on the performance of software applications have already appeared [2,13,17–20,23]. Some of them analyze different forms of code mobility in "isolation" [17,23], independently of their utilization within particular applications, thus providing some form of general guidelines that can help in taking decisions during the design of an application. Other approaches consider particular applications that exploit code mobility [2,13,18–20], and evaluate their performance using "ad hoc" models. On the contrary, our work is addressed to the definition of a general methodology that can be used as a support for the designer during the early phases of the software development, providing insights into the consequences of architectural choices that can be taken during those phases.

The paper is organized as follows. In Section 2, we present a formalism for the specification of an SA that encompasses both the client–server and the mobile code styles. In Section 3, we first informally present a simple "context-dependent" software application [5] that will be used throughout the paper as an example to show the steps of our methodology, and then, as a first step, we give its specification using the formalism introduced in Section 2. In Section 4, we present our performance evaluation methodology, showing how an MDP can be derived from the SA specification and how to solve it to get performance results; at the end of Section 4 we present the results of the solution of the MDP obtained for our example application, showing the kind of insights we can derive from our methodology. In Section 5, we make some final considerations.

2. Mobility-based SA description

The starting point of our methodology is the description of a software application in a suitable ADL that includes the possibility of specifying components location, as well as component mobility. Such languages have been proposed quite recently [6,9,18,26]. Our approach is not tied, in principle, to a particular language. However, for the sake of examples, we adopt a modification of the COMMUNITY language [26]. Besides some

slight syntactic changes, the main modification is the introduction of a new connector type, as described below.

In this ADL, an architecture description consists of the following sections:

- *Components-type*, that defines the type of the architecture components;
- *Connectors-type*, that defines the type of the architecture connectors;
- *Components*, that defines the actual instances of the architecture components;
- *Connections*, that defines the actual instances of the architecture connectors.

A component type consists of a set of *named guarded* actions that can modify only local variables, where each action consists of the simultaneous atomic execution of assignment statements. An action is chosen (non-deterministically) for execution if its guard is true. The component type is specified as follows (we refer to [26] for a detailed definition):

program $P(\lambda)$
var V
init *Icond*
do $\underset{g\in C}{[\,]}\ g : \left[B(g) \rightarrow \underset{a\in D(g)}{\|}\ a := F(g,a)\right]$

where P is the component type *name*, V is the set of *local variables*, λ is a parameter indicating component location, *Icond* is a proposition that states the initial values of V, G is a set of *action names*, $B(g)$ is the *guard* associated to the action named g, $D(g)$ is the subset of V that the action named g can modify, and $F(g,a)$ expresses the value to be assigned to a when g is executed. At the time of component instantiation an initial value can be assigned to the location parameter λ that can be modified at "run-time".

A connector type defines a pattern of interaction among components. In COMMUNITY, some connector types are introduced. In particular, we focus on the *Communicator* connector type that models synchronous message sending between two components. Its prototype is as follows: [1]

connector *Communicator*(c1, c2: **program**; a1, a2: **action**; x1, x2: **any_type**; I: **condition**)

In this definition c1 and c2 are the names of the connected components, a1 and a2 are the names of actions performed by c1 and c2, respectively, "synchronized" by the connector, while x1 and x2 are c1 and c2 variables, respectively, used to send and receive the exchanged value; I is a condition that controls the connector activation. The semantics of this connector is such that action a1, that starts communication, is executed only when both its guard and I hold true. A communication is completed when the guard of action a2 holds true. In practice, the guards of action a1 and a2 model the willingness of the two partners to participate in the communication, while I can be used to model some condition that actually enables communication but that is non-local to any component (e.g., modeling the concept of two components' co-location). The *Communicator* formal semantics, specified in [26], guarantees that the "receive" action a2 is executed only after the communication takes place, i.e., after the transfer of the value of x1 to x2 has been completed.

Based on this connector type, we define an additional one, called *Mob_Comm*, that can also model code mobility; its prototype is as follows:

[1] For our purposes, it suffices to give the connector prototype and an informal description of its semantics. Its "body", formally specifying its semantics, can be expressed by a set of guarded statements, analogously to a component type; see [26] for details.

connector *Mob_Comm*(c1, c2: **program**; a1, a2: **action**; x1, x2: **any_type**; I: **condition**; M: **mob_condition**)

All the common parameters play the same role in both connectors. In addition, the parameter *M*, which differentiates *Mob_Comm* from *Communicator*, plays a special role, since it allows us to express the mobility of the connected components. It carries a "mob_condition" type because, besides the Boolean values of **true** and **false** of a standard condition, it can also be instantiated with a special value "**?**", with the following semantics:

- when *M* is instantiated with a **false** value (in general, an expression yielding this value), *Mob_Comm* has exactly the same semantics as *Communicator*, i.e., synchronous message sending from c1 to c2 component.
- when *M* is instantiated with a **true** value, then the connector semantics is modified as follows with respect to *Communicator*: the activation rule is the same as *Communicator* (i.e., controlled by I condition), but when the connector is activated the location of c1 changes to that of c2, before transferring the value of x1 to x2 according to the *Communicator* semantics.
- when *M* is instantiated with a "**?**", the connector semantics corresponds to the non-deterministic execution of both the above options; this is helpful in generating a model where the mobility policy must not necessarily be chosen in advance, but can result (as the optimal one) from the model solution.

The *Components* and *Connectors* sections instantiate the actual components and connectors of the system SA, by specifying their names and the corresponding types, and the value of their parameters.

The execution model of an application described by this language can be operationally viewed as a non-deterministic fair interleaved selection of actions in the **do** sections of all the instantiated components that start from the initial state described by the union of all the **init** sections. At each execution step, only actions whose guards hold true are considered for execution.

The ADL does not intend to be a "complete" language for the specification of mobility-based SAs. Indeed, other connector types would likely be necessary, since *Mob_Comm* appears suited to model a *mobile agent* style, whereas it is less clear whether it can model other kinds of mobile code paradigms as well [11]. Moreover, even in the mobile agent style, the destination of an agent does not necessarily coincide, in general, with the location of its communication partner. However, this ADL is sufficient to give an idea of the features that an ADL for mobility-based SA should have, and it represents the starting point of our evaluation methodology that, as remarked in the introduction, is our main focus.

With regard to the choice of expressing component mobility within a connector, we would like to remark that this is not the only possible choice. For example, in [26] mobility is expressed within the body of a component, as an assignment statement that changes the value of the component location variable. According to the non-deterministic operational semantics outlined above, this choice implies that a mobile component could potentially consider the possibility of changing location at each execution step. On the contrary, our choice limits the points where a voluntary location change can happen to the points where a *Mon_Comm* connector is active (i.e., an interaction is taking place). We adopt this choice mainly because it allows us to derive a simpler model for the analysis of the application, as it will become evident in the following. As a positive "side-effect", this choice reinforces the idea that mobility should be considered as an interaction relation option, in the sense that no component should voluntarily move just for the sake of moving.

Rather, it should move when this may be advantageous, and from a performance viewpoint the possible advantages are strictly related to interactions. On the other hand, the constraint imposed by our choice should not adversely affect the application efficiency, since the impact of mobility on performance manifests itself mostly when an interaction takes place.

3. Application example

In this section we present the application we use as example throughout the paper. After an informal description, we formally specify its SA using the ADL introduced in Section 2.

The application we consider is a simple example of *context-dependent* application [5], i.e., an application that is aware of its context and context changes, and that accordingly modifies its behavior. We consider only the "skeleton" of such an application, abstracting from several details.

The computing platform for such an application consists of a portable device (PD) with wireless connections, and a server (SV) connected to a fixed network. In our example, the application "context" simply consists of the physical position of PD; when PD is within a building, information about its position is managed by a component (AB) located at SV, while it is managed by a component (GPS) [2] located at PD, when PD is outdoor. The application "core" consists of a monitor (MON) that periodically checks the current PD position (depending on the position value, or its change, this check could trigger opportune actions). In order to check this position, MON must communicate with AB, when PD is indoor, and with GPS when PD is outdoor. MON may be located at both PD or SV; the two solutions may have different impacts on the interaction cost, depending, for instance, on how long PD is in indoor or outdoor positions.

Note that in the overall application we have two software components (AB and GPS) whose locations are fixed, and one component (MON) whose location needs further investigation. Indeed, it is not a priori clear whether MON would be better located at PD or SV, from a performance viewpoint, and whether its location should be fixed or should change dynamically. In other words, whether a client–server or mobile agent style is better suited to design the interactions among MON, AB and GPS.

Moreover, note also that in this application, besides the potential presence of code mobility (so-called *logical mobility*), we also have *physical mobility* of the PD device.

3.1. Modeling the logical mobility

The SA of this application can be described as follows:

Components-type
program mon(λ)
var sent, received, req: **bool**; oldpos, newpos: position;
init sent = **false and** received = **true and** newpos = **null and** req = **false**
do send_req: [sent = **false** —> sent := **true** ‖ oldpos := newpos ‖ req := **true**]

[2] The acronyms AB and GPS for the components that manage location data are not casual, but intend to recall the ActiveBadge [25] and Global Positioning System [10] positioning technologies.

[ ] get_pos: [**true** —> received := **true** ‖ sent := **false** ‖ *Check*(oldpos, newpos) ‖ req := **false**]

program ab(λ)
var pos: position; req: **bool**;
init req = false
do wait_req: [**true** —> pos := *Current_pos*()]
[ ] send_pos: [req = **true** —> req := **false**]

program gps(λ)
var pos: position; req : **bool**;
init req = false
do wait_req: [**true** —> pos := *Current_pos*()]
[ ] send_ pos: [req = **true** —> req := **false**]

Connectors-type
connector *Mob_Comm*(c1, c2: **program**; a1, a2: **action**; x1, x2: **any_type**; I: **condition**; M: **mob_condition**)

Components
MON : mon(PD); AB : ab(SV); GPS : gps(PD);

Connections
mon-to-ab: *Mob_Comm*(MON, AB, MON.send_req, AB.wait_req, MON.req, AB.req, *ABactive*(), **?**)
ab-to-mon: *Mob_Comm*(AB, MON, AB.send_pos, MON.get_pos, AB.pos, MON.newpos, *ABactive*(), **false**)
mon-to-gps: *Mob_Comm*(MON, GPS, MON.send_req, GPS.wait_req, MON.req, GPS.req, *GPSactive*(), **?**)
gps-to-mon: *Mob_Comm*(GPS, MON, GPS.send_pos, MON.get_pos, GPS.pos, MON.newpos, *GPSactive*(), **false**)

Auxiliary functions
bool *GPSactive*() {PD.λ = OUTDOOR}
bool *ABactive*() {PD.λ = INDOOR}

The *Components-type* section describes the types of the involved components; as it can be seen, in the program body we only describe interaction-related behavior. A "mon" program sends a position request and, after receiving the response, compares the previous and current positions (*Check*(·, ·): this could trigger other actions not considered here); on the other hand, the "ab" and "gps" programs behave analogously: they wait for a position request, and when it arrives, they set the position (*Current_pos*()) and reply with the appropriate response.

The *Connectors-type* section includes the only connector type used in this example, i.e., *Mob_Comm*.

The *Components* section defines the actual component instances; in this example we have one instance for each component type. For each instance, the initial value of the corresponding location variable λ is also specified.

Finally, the *Connections* section defines the actual connections existing among the application components. All the shown connections are instances of the *Mob_Comm* connector. "mon-to-ab" and "mon-to-gps" model the message sending from a "mon" program to

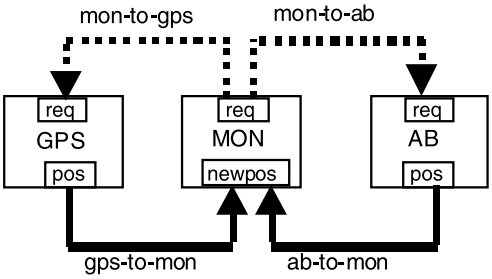

Fig. 1. SA of the context-dependent application.

"ab" and "gps" programs, respectively, while "ab-to-mon" and "gps-to-mon" model the corresponding replies.

The *ABactive*() and *GPSactive*() boolean functions are used to instantiate the connector condition I, that controls the actual connection activation; in this case it holds true only when the interaction is meaningful, i.e., the involved program (either "ab" or "gps") is able to give correct information about the current PD position.

With regard to the condition M, that controls the components mobility during interactions, we can see that it is set to false in "ab-to-mon" and "gps-to-mon" connectors, since our design choice is such that a "gps" or "ab" component can never change the location, while it is set to "**?**" in "mon-to-ab" and "mon-to-gps", since we retain the location and the possible mobility of "mon" component yet under investigation at this design stage.

A graphical description of this SA (its "box-and-line" diagram) is shown in Fig. 1. In this figure continuous arrows represent the *Mob_Comm* connector with M condition set to false, while dashed arrows represent the *Mob_Comm* connector with M condition set to "**?**". As a consequence, the (possibly) mobile components are evidently those from which at least a dashed arrow starts. In our example, only MON is such a component.

3.2. Adding physical mobility

As said before, in our example also computing devices may be mobile, in a *physical* sense that is different from code *logical* mobility sense. These two kinds of mobility obviously represent different kinds of behavior in the real world. Anyway, at an appropriate abstraction level, they can also be considered as different "instantiations" of a unique mobility model, as recognized in several papers [6,18,26]. For this reason, and also to have a homogeneous description of the overall application that facilitates the following generation of a performance evalution model, we generalize the adopted ADL to describe both the software components and hardware devices that support them. In this generalization, a hardware device is described using the same syntax as a software component. A consequence of this generalization is that a component location variable can take as a value the name of another component. Basically, this is used to model a "software" component hosted by a "hardware" component. In general, using component names as possible values of the location variables also allows us to model nested software components. In terms of mobility, this allows us to easily model the relocation of a component as a whole (i.e., jointly to all the possibly nested components), without explicitly modifying the location of the nested components. Fig. 2 shows this effect in our example, where the possible (physical, in this case) location change of PD from OUTDOOR to INDOOR brings the side-effect of (physical) location change of the nested GPS and MON components from

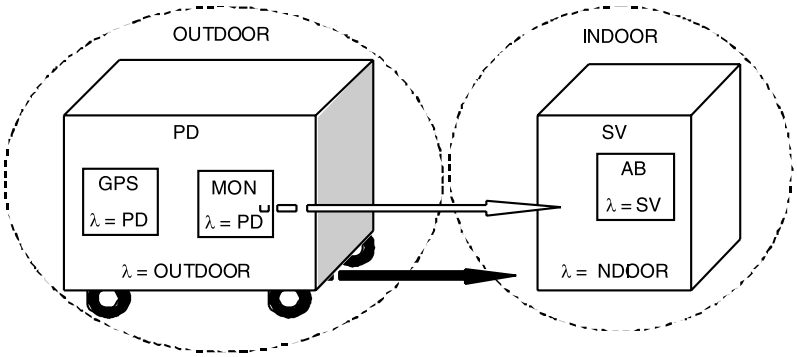

Fig. 2. Example of mobility of "nested" components.

OUTDOOR to INDOOR as well, whereas the possible location change of MON from PD to SV does not bring any side-effect, since MON does not have any nested component.

Physical connections among hardware components can be represented introducing appropriate connectors; in our example, we introduce a new connector type, whose prototype is defined as follows:

connector *Wireless_link*(c1, c2 : **program**)

This connector type models, in a simple way, the existence of a bidirectional wireless link between component c1 and c2. It suffices for our example scope, whereas in a general case additional information could be necessary (for example, the connector body could contain local variables that describe the link "quality", like its bandwidth, and statements modeling variations of this quality over time).

With the generalization described above, the description of the example application becomes as follows (for the sake of conciseness, we omit the bodies of already introduced software components):

Components-type

program mon(λ)
. . .
program ab(λ)
. . .
program gps(λ)
. . .
program pd(λ)
init
do in_to_out: [λ = OUTDOOR —> λ := INDOOR]
[ ] out_to_in: [λ = INDOOR —> λ := OUTDOOR]

program sv(λ)
init
do

Connectors-type

connector *Mob_Comm*(c1, c2: **program**; a1, a2: **action;** x1, x2: **any_type**;
I: **condition**; M: **mob_condition**)
connector *Wireless_link*(c1, c2: **program**)

Components

MON : mon(PD); AB : ab(SV); GPS : gps(PD); SV : sv(INDOOR); PD : pd(INDOOR);

Connections

mon-to-ab: *Mob_Comm*(MON, AB, MON.send_req, AB.wait_req, MON.req, AB.req, *ABactive*(), **?**)
ab-to-mon: *Mob_Comm*(AB, MON, AB.send_pos, MON.get_pos, AB.pos, MON.newpos, *ABactive*(), **false**)
mon-to-gps: *Mob_Comm*(MON, GPS, MON.send_req, GPS.wait_req, MON.req, AB.req, *GPSactive*(), **?**)
gps-to-mon: *Mob_Comm*(GPS, MON, GPS.send_pos, MON.get_pos, GPS.pos, MON.newpos, *GPSactive*(), **false**)
sv–pd: *Wireless_link*(SV, PD)

Auxiliary functions

bool *GPSactive*() {PD.λ = OUTDOOR}
bool *ABactive*() {PD.λ = INDOOR}

As we can see, this new description of the application consists of two more components (SV and PD) and one more connection (sv–pd). SV is an instance of the "sv" component type; in our example, we do not need to specify information about this component, apart from its physical location, which is initialized to INDOOR as the actual location parameter of the instance, and does not change over time; hence, the **init** and **do** sections of its body are empty. PD is an instantiation of the "pd" component type; in this case, its initial location is set to INDOOR (but it could be OUTDOOR as well), while, to model its mobility, its body includes two statements that alternatively set the location to INDOOR or OUTDOOR.

It is worth remarking the "operational" difference between the mobility expressed inside the body of a *Component-type* specification (e.g., the "pd" program case) and the one expressed in the semantics of a *Connector-type* specification (the *Mob_Comm* connector case). The former can be considered as an explicit willingness of the component itself to move independently of the next interaction the component will be involved in. The latter represents a kind of external command given to the component for moving, that depends on non-local conditions, due to the specific features and conditions of the interaction the component is going to be involved in.

Finally, from the actual values assigned to the location parameters λ in the components section, we can note that the "locations" used in this example are four (PD, SV, INDOOR, OUTDOOR) where the former two are actually hardware devices, while the latter two are "geographical" locations.

4. Performance evaluation methodology

The idea behind our methodology is that the opportunity (from a performance viewpoint) of the adoption of a mobile code architectural style with respect to a client–server one may be better investigated by evaluating whether it improves some performance index of interest. To this purpose, we should consider that an SA describes a system at a very abstract level, with many details about internal components behavior left undefined. Most of the information that we can extract at this stage concern interactions among components. As a consequence, we focus in this paper on performance indices that can be expressed, in

general, as functions of the number and types of interactions among components. Several relevant performance indices for distributed applications belong to this last category. This is the case of cumulative measures like the totally generated network traffic, possibly limited to some connections (e.g., over wireless links), or the energy consumption of battery powered devices, where the dependence on the interactions derives from the consideration that the energy consumption for wireless communications represents a significant fraction of the overall energy consumption of a portable device [14, 22]. The two mentioned indices are of special interest for applications that are (partly) supported by portable devices with wireless connections, since in this case both wireless bandwidth and energy are scarce resources whose utilizations deserve attention.

We present an evaluation methodology restricted to this kind of indices (i.e., cumulative measures related to interactions), and whose goal is to provide insights into the opportunity of adopting mobility policies. To this purpose we build a stochastic model that describes the system dynamics, starting from the description of the system SA in the ADL presented in Section 2. The construction of the model is completely automatic, except for the assignment of probabilities to state transitions that requires human intervention. This "manual" step is unavoidable in every performance evaluation approach that does not exploit data coming from model execution or from a previous usage of similar software. Being one of our claims the possibility of *early* effectiveness evaluation, this manual effort may be fairly alleviated by extracting part of this information (e.g., frequencies of different actions) directly from the system specifications.

The stochastic model we build is an MDP [24]. We recall that an MDP is defined by a tuple $\langle S, A, p, r\rangle$, where S is a set of *states*, A is a set of *decisions, p* is a *transition function* p: $S \times A \times S \to [0, 1]$ that defines the one-step state transition probabilities, and r is a *reward function* $r : S \times A \to \Re$, with $\Re$ the set of real numbers, that defines the average reward gained each time a state is visited. As we can see, both the transition probabilities and the reward in each state are decision dependent. A *policy* is a function $\sigma : S \to A$ that selects the decision to be taken in each state (and hence the corresponding reward and transition probabilities). Using results from MDP theory it is possible to determine the policy that optimizes measures like the reward accumulated in steady-state conditions.

In the MDP that we derive from the SA description, we model the choice between the mobility and no-mobility option as the choice between alternative decisions, while the reward associated to each state is related to the performance index we are interested in. In this framework, we can get insights into the opportunity of adopting a mobile code paradigm by determining the optimal policy that minimizes the steady-state accumulated reward: if the optimal policy corresponds to choosing the mobility option in some states, this can be considered both as an indication that it is worth considering the adoption of mobile code style in the system development and as an early indication of which mobility policy is worth adopting.

The process of deriving and solving the MDP from the SA description is presented in the following. It consists of three steps:

- [ADL]-to-[labeled graph],
- [labeled graph]-to-[MDP],
- [MDP simplification and solution].

For each step, we give in the following subsections the input and output data, the procedure to be followed, and a comment that explains and motivates the operations performed in that step. Moreover, we apply each step to the example application presented in Section 3.

4.1. A model of the application dynamics

Step: [ADL]-to-[labeled graph]
Input:

- the specification of the system SA in the given ADL;
- the function $f(\cdot)$ that defines the "cost" associated to the execution of a given system action.

Output: the description of the system dynamic behavior, given by a *labeled directed graph* $\mathrm{LG} = \langle N, T, L_T \rangle$, where

- N is a set of *nodes*, with each node $s \in N$ representing a possible system state (i.e., values of the local variables of components and connectors);
- $T \subseteq N \times N$ is a set of directed arcs, with each arc representing a transition caused by a system action;
- $L_T : T \rightarrow \{\textit{arc labels}\}$ is an arc labeling function, where an *arc label* is defined as a pair (*action name, cost*).

The goal of this step is to build a "graphical representation" of the application dynamics. An interaction cost can change dynamically, due to location changes. Hence, the static description of an SA (its "box and line" diagram, as depicted in Fig. 1) does not contain enough information to perform an accurate evaluation of interaction related attributes. To this purpose, we derive from the SA description a labeled graph LG, i.e., a graph where each node represents the execution a given application state (e.g., component locations and internal state), and each arc represents the execution of a given action in the system (e.g., the transfer of a value during an interaction). Each arc label contains information about the cost of the corresponding action (e.g., interactions between components that share the same location can be considered as having a negligible cost with respect to interactions involving communications through the network). To this purpose we introduce the cost function $f(\cdot)$ associated to each system action. This cost must fairly be related to the performance index we want to evaluate. For example, if the performance index we are interested in is the total network traffic over wireless links, then the cost of an action that transfers a value x from a component c1 to a component c2 could be *sizeof*(x) if c1 and c2 communicate through a wireless link, and 0 otherwise.

We recall that we have introduced in our ADL the *Mob_Comm* connector, where a condition M controls the choice between two options (mobility and no-mobility). If M holds either **true** or **false** when the connector is activated, then only one of the two options is selected; as a consequence, only the corresponding arc appears in LG. Otherwise, if M is set to "**?**", then both options are non-deterministically selected, and hence both the arcs corresponding to the two options belong to T, each labeled with the corresponding cost. We call this particular pair of arcs a pair of *conjugate* arcs, and the node these arcs start from a *decisional* node. Note that decisional nodes are likely to be a small subset of the nodes of LG, due to the embedding of the choice of a mobility policy in the *connector-type* rather than in the body of the *Component-type*, that limits decisional nodes to those nodes where a *Mob_Comm* connector with $M =$ "**?**" can be activated. Instead, by embedding the mobility in the body of a component, a larger set of nodes may play the role of decisional nodes, since mobility would not be constrained to take place only when a communication occurs.

The derivation of LG from the SA description may be automatically performed, e.g. using a transition system that expresses the operational semantics corresponding to the execution model outlined in the previous section. It is out of this papers scope to give a

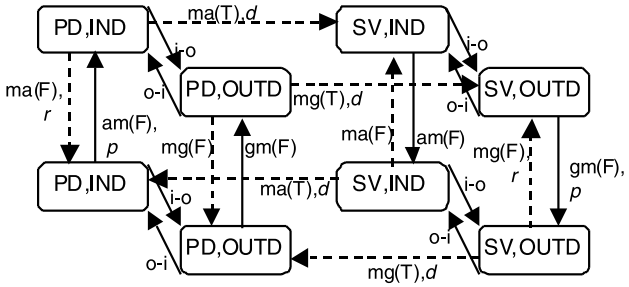

Fig. 3. Labeled graph of the context-dependent application.

set of formal rules that express this semantics. Indeed the starting point for the application of our methodology is LG, that can be considered as a graphical representation of the transition system.

In addition to the ADL that we have defined, any ADL from which an analogous transition system can be obtained is in principle suited to specifiy the system SA. Of course this ADL must include the notion of location and movement to different locations.

Fig. 3 shows the labeled graph for our example. A pair of dashed arrows starting from a node represents the pair of conjugate arcs corresponding to the activation of the *Mob_Comm* connector with control condition set to "**?**".

For the sake of clarity, we show in each state the location of MON (either PD or SV) and of PD (IND for indoor or OUTD for outdoor). Besides each arc we show the arc label, that is a pair where the first element (in plain text) indicates the corresponding system action, and the second element (in italics) the corresponding value of the cost function. The acronyms used in Fig. 3 for the system actions are explained in Table 1. As cost measure we adopt the number of bytes transmitted over a wireless link, where r is the size of a request from MON, p is the size of a reply from GPS or AB, and d is the size of MON (when it changes location); wherever the second element of the pair is not shown it corresponds to a cost equal to zero. Note that a zero cost may correspond either to a system action that does not involve any communication, or to a system action that involves communication between co-located components (with the implicit assumption that co-located components can always communicate with null costs). In our example, the former case corresponds to the physical movement of PD, and the latter, for instance, to a communication between MON and GPS when the location of PD is OUTDOOR, and the location of MON is PD.

Note also that the SA architecture description given in Section 2 contains all the information needed to automatically derive the value of the arc labels. Indeed, for each action we know whether it involves communication between components (i.e., involves the activation

Table 1
Acronyms used in Fig. 2 for the system actions

Acronym	Corresponding system action
i–o	Activation of in-to-out action of PD component
o–i	Activation of out-to-in action of PD component
ma(T) (ma(F))	Activation of mon-to-ab connector with M = true (false)
am(F)	Activation of ab-to-mon connector with M = false
mg(T) (mg(F))	Activation of mon-to-gps connector with M = true (false)
gm(F)	Activation of gps-to-mon connector with M = false

of *Mob_Comm* connector), and for each communication we know whether it involves the use of the *Wireless_link* connector (depending on the components "physical" location). In Appendix A we show how we can formally define the cost function $f(\cdot)$ for the considered example, using information derived from the SA description, in the case where the performance measure we are interested in is the traffic over wireless links.

4.2. MDP construction[3]

Step: [labeled graph]-to-[MDP]
Input: the labeled graph LG $= \langle N, T, L_T \rangle$ produced in the first step.
Output: an MDP $P = \langle S, A, p, r \rangle$, obtained as follows:

- $S = N$;
- $A = \{n, m, no\_m\}$ (that stands for null decision, mobility and no-mobility, respectively);
- the enabled decisions in each state, and the transition probability and reward functions are defined as follows, by distinguishing the case of states corresponding to non-decisional or decisional nodes of LG:
  1. for each state s of P corresponding to a non-decisional node of LG:
     - the set of enabled decisions of s is $A(s) = \{n\}$;
     - the transition probability and transition reward functions are defined in Tables 2 and 3, respectively, where $Succ(s) = \{s' | (s, s') \in T\}$, and $cost(s, s')$ is the second element of the pair $L_T(s, s')$. The assigned transition probabilities must satisfy the constraint $\sum_{s' \in Succ(s)} p(s, n, s') = 1$.
  2. for each state s of the MDP corresponding to a decisional node of LG:
     - the set of enabled decisions of s is $A(s) = \{m, no\_m\}$;
     - the transition probability and transition reward functions are defined in Tables 2 and 3, respectively, where $Succ(s)$ and $cost(s, s')$ have been defined above. The assigned transition probabilities must satisfy the constraints:

$$\sum_{s' \in Succ(s) - s_{\text{true}}} p(s, no\_m, s') = 1, \tag{1}$$

$$\sum_{s' \in Succ(s) - s_{\text{false}}} p(s, m, s') = 1, \tag{2}$$

     where $s_{\text{false}} \in Succ(s)$ and $s_{\text{true}} \in Succ(s)$ denote the two states reached by the conjugate arcs (in LG) starting from s, and corresponding to the activation of the *Mob_Comm* connector with condition M set to **false** and **true**, respectively.

LG can be viewed as the "skeleton" used for building the actual evaluation model. The outgoing arcs from a given node of LG model the next action to be executed in a given system state, non-deterministically chosen according to the semantics underlying LG. To carry out an evaluation in a stochastic setting, we must turn this non-deterministic choice into a probabilistic one. Hence, while the first step is completely automatic, in this second step human intervention is necessary to guide the process of attaching opportune probabilities to these transitions (to this purpose, action names that label each arc

[3] This step is here described under the hypothesis of at most one pair of conjugate arcs leaving a decisional node, but it can be easily extended to the general case of whatever finite number of conjugate pairs per decisional node.

Table 2
Transition probability function of the MDP

$p(s,a,s') =$"user defined"	if $a \in A(s) \wedge s' \in Succ(s)$
$p(s,a,s') = 0$	if $a \in A(s) \wedge s' \notin Succ(s)$
$p(s,a,s') =$ undefined	if $a \notin A(s)$

Table 3
Average reward function of the MDP

$r(s,a,s') = cost(s,s')$	if $a \in A(s) \wedge s' \in Succ(s)$
$r(s,a,s') =$ undefined	if $a \notin A(s)$

of LG may help). The first row of Table 2 contains all the quantities that need a human estimate.

A special kind of non-determinism is modeled in LG by decisional nodes and the pair of conjugate arcs starting from each of them. These arcs represent the "activation" of *Mob_Comm* connectors with Boolean condition *M* set to "**?**". In the semantics of our ADL, this indicates a non-deterministic choice between the two options of mobility and no-mobility. From a design viewpoint, this indicates that we do not have enough information for choosing between the two options, i.e., for turning the non-deterministic choice into a deterministic one. As remarked at the beginning of this section, we embed in an MDP setting the uncertainty about the two options, by mapping alternative options in a given node of LG to alternative decisions in the corresponding state of the MDP. Choosing one option in each decisional state corresponds to determining a given policy σ in the MDP. Hence, we can use results from MDP theory to solve the MDP obtained from LG and determine the optimal policy that minimizes the accumulated cost. This policy states which option (mobility/no-mobility) should be chosen in all the states where this is possible (decisional states). Hence, it can give us suggestions about the opportunity of implementing code mobility in the successive development phases, and about the mobility requirements that the application should satisfy to exploit the advantage of this paradigm (i.e., to improve the performance metrics of interest).

Note that the reward function that we define is related to transitions from state to state. The average reward gained each time a state is visited is given by the sum of the rewards for all outgoing transitions from this state, weighted with their probabilities. Hence, for non-decisional states the average reward in state s is:

$$r(s,n) = \sum_{s' \in Succ(s)} p(s,n,s') r(s,n,s') \tag{3}$$

while for decisional states the average reward in state s is

$$r(s,a) = \begin{cases} \sum_{s' \in Succ(s) - s_{\text{false}}} p(s,m,s') r(s,m,s') & \text{if } a = m, \\ \sum_{s' \in Succ(s) - s_{\text{true}}} p(s,no\_m,s') r(s,no\_m,s') & \text{if } a = no\_m. \end{cases} \tag{4}$$

Fig. 4 shows the MDP obtained from the labeled graph of Fig. 3. For the sake of simplicity, we have substituted in this figure the node labels with numerical labels. For each state i, each outgoing transition toward state j is labeled with the value of the associated reward and transition probability. In this example, the user defined values (denoted by v1, v2, ..., v20) of the transition probability function are assigned as follows:

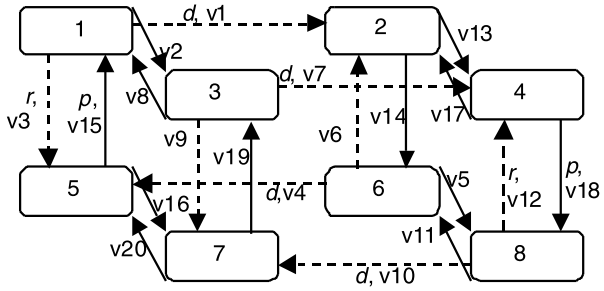

Fig. 4. MDP of the context-dependent application.

decisional states:

$p(1, m, 2) = \text{v1}$
$p(1, m, 3) = p(1, no\_m, 3) = \text{v2}$
$p(1, no\_m, 5) = \text{v3}$
$p(6, m, 5) = \text{v4}$
$p(6, m, 8) = p(6, no\_m, 8) = \text{v5}$
$p(6, no\_m, 2) = \text{v6}$
$p(3, m, 4) = \text{v7}$
$p(3, m, 1) = p(1, no\_m, 1) = \text{v8}$
$p(3, no\_m, 7) = \text{v9}$
$p(8, m, 7) = \text{v10}$
$p(8, m, 6) = p(8, no\_m, 6) = \text{v11}$
$p(8, no\_m, 4) = \text{v12}$

non-decisional states:

$p(2, n, 4) = \text{v13}$
$p(2, n, 6) = \text{v14}$
$p(5, n, 1) = \text{v15}$
$p(5, n, 7) = \text{v16}$
$p(4, n, 2) = \text{v17}$
$p(4, n, 8) = \text{v18}$
$p(7, n, 3) = \text{v19}$
$p(7, n, 5) = \text{v20}$

We would like to remark that the choice of assigning the same probability value to transitions toward the same state under different decisions (e.g., $p(1, m, 3) = p(1, no\_m, 3) = \text{v2}$) is relative to this example only, and should not be intended as a general rule. The only constraints on the transition probabilities are those given by relations (1) and (2), so that, for example, it can be in general $p(1, m, 3) \neq p(1, no\_m, 3)$. As in Fig. 2, where the reward is omitted it is equal to zero.

4.3. Model solution

Step: MDP simplification and solution
Input: MDP $P = \langle S, A, p, r\rangle$ built at the second step.
Output: Optimal policy obtained solving a "simplified" MDP $= \langle \hat{S}, \hat{A}, \hat{p}, \hat{r}\rangle$, obtained as follows:

Let $S_0 \subseteq S$ be the subset of all the non-decisional states of P whose outgoing arcs have cost equal to zero, and let $S_N = S - S_0$.
Let $\mathbf{P}_{0N} = [p_{0N}(s', s'')]$ be an $|S_0| \times |S_N|$ matrix, with $s' \in S_0, s'' \in S_N$ and $p_{0N}(s', s'') = p(s', n, s'')$.
Let $\mathbf{P}_{00} = [p_{00}(s', s'')]$ be an $|S_0| \times |S_0|$ matrix, with $s' \in S_0, s'' \in S_0$ and $p_{00}(s', s'') = p(s', n, s'')$.
Let $\mathbf{H} = [h(s', s'')]$ be an $|S_0| \times |S_N|$ matrix, with $s' \in S_0, s'' \in S_N$ and $\mathbf{H} = (\mathbf{I} - \mathbf{P}_{00})^{-1}\mathbf{P}_{0N}$.
Then, we have:

- $S = S_N$;
- $\hat{A} = A$;

- $\hat{p}(s,a,s') = p(s,a,s') + \sum_{s'' \in S_0} p(s,a,s'')h(s'',s')$, with $s \in \hat{S}$, $s' \in \hat{S}$, $a \in A(s)$;
- $\hat{r}(s,a) = r(s,a)$, with $s \in \hat{S}$, $a \in A(s)$, where $r(s,a)$ is defined by (3) and (4).

Solving an MDP means finding a *policy* that selects a decision in each state, so that the total accumulated reward is minimized. In our perspective, if the obtained optimal policy selects in some states the mobility option, this can be considered as an indication that code mobility may represent an effective style for the considered application, and the optimal policy can be adopted as a mobility strategy (even if it can be later modified due to further design insights). To alleviate the computational problems caused by the state explosion, we simplify the process obtained in step 2, before solving it, with the aim of reducing the size of its state space. To this purpose, we drop all the non-decisional states whose outgoing arcs have cost equal to zero, and modify transition probabilities among the remaining states so that the new process is equivalent to the original one, in the sense that the policy that optimizes the original MDP also optimizes the reduced one (see Appendix B for a proof). This simplification can be automatically performed, according to the algorithm outlined above, and in practice it can be quite effective, since it is generally likely that several transitions of the original process have cost equal to zero (for example, this is the case when we are interested in costs related to wireless communications, and the software components are hosted by nodes connected by wireless and wired links). Fig. 5 shows the simplified MDP for our example. For each state *i*, each outgoing transition toward state *j* is labeled with the value of its probability (denoted by q*k*), whereas the reward value is shown within the state, besides the numerical label *i* that identifies the state. Both transition probabilities and reward values have been obtained applying the simplification algorithm to the MDP shown in Fig. 4. Note that within decisional states a pair of reward values is shown, where the first element is the reward associated to the *no_m* decision, while the second element is the reward associated to the *m* decision. Table 4 shows the transition probabilities q*k*, as functions of the transition probabilities values v*h* of the original MDP.

In general a symbolic solution of the process would be suitable, but it is not always feasible. Anyway, the process can be numerically solved [24] by instantiating appropriately the model parameters, where each instantiation defines a possible implementation scenario, so that we can compare the impact of code mobility on different scenarios. Note that the complexity of the solution depends on both the size of the state space, and the number of decisional states. Hence, the reduction in the number of decisional states obtained by

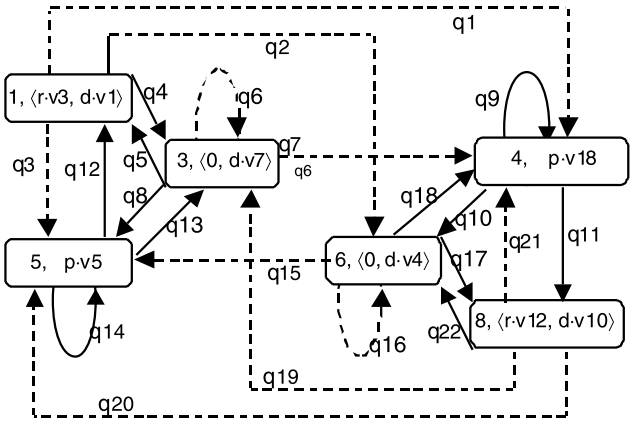

Fig. 5. Simplified MDP.

Table 4
Transition probabilities of the simplified MDP, expressed in terms of transition probabilities of the original MDP

$q1 = v1 \cdot v13$	$q12 = v15$
$q2 = v1 \cdot v14$	$q13 = v16 \cdot v19$
$q3 = v3$	$q14 = v16 \cdot v20$
$q4 = v2$	$q15 = v4$
$q5 = v8$	$q16 = v6 \cdot v14$
$q6 = v9 \cdot v19$	$q17 = v5$
$q7 = v7$	$q18 = v6 \cdot v13$
$q8 = v9 \cdot v20$	$q19 = v10 \cdot v19$
$q9 = v17 \cdot v13$	$q20 = v10 \cdot v20$
$q10 = v17 \cdot v14$	$q21 = v12$
$q11 = v18$	$q22 = v11$

encoding the mobility in a connector signature (by means of condition M), as outlined in Sections 2 and 3, has a positive impact on the complexity of solution.

In our example, we have considered several scenarios by varying the model parameters that are transition probabilities $p(s, a, s')$, and costs of system actions (see Appendix A) that determine the values of the average reward function. The basic actions that can be performed in our model (and that make the system changing state) are: *message sending* (i.e., either position request or position value reply) and *change of physical location* (i.e., moving either from indoor to outdoor or vice versa). It can be properly assumed that the transition probabilities are based on the relative frequencies of these actions. The frequency of message sending actions does not depend on their type, since a position value reply always takes place upon a position request. We can therefore assign the same frequency to these two actions. The frequency of location changes can instead differ, depending on the "directions" of the transit. Besides, message sending is a type of action that is typically more frequent than location change, so the frequency of message sending can be considered as the unit to which relating the frequency of location change, in the following way:

freq(position request) = freq(position value reply) = 1
freq(indoor to outdoor) = $1/n_{\mathrm{io}} = f_{\mathrm{io}}$
freq(outdoor to indoor) = $1/n_{\mathrm{oi}} = f_{\mathrm{oi}}$

where n_{io} (n_{oi}) is the average number of message sending occurring between two transits from indoor to outdoor (from outdoor to indoor). For example, $f_{\mathrm{io}} = 0.1$ means that the transit from indoor to outdoor position occurs once every 10 message sending. The transition probabilities are therefore given by the following relations:

decisional states:

$p(1, m, 2) = \mathrm{v1} = 1/(1 + f_{\mathrm{io}})$
$p(1, m, 3) = p(1, no\_m, 3) = \mathrm{v2} = f_{\mathrm{io}}/(1 + f_{\mathrm{io}})$
$p(1, no\_m, 5) = \mathrm{v3} = 1/(1 + f_{\mathrm{io}})$
$p(6, m, 5) = \mathrm{v4} = 1(1 + f_{\mathrm{io}})$
$p(6, m, 8) = p(6, no\_m, 8) = \mathrm{v5} = f_{\mathrm{io}}/(1 + f_{\mathrm{io}})$
$p(6, no\_m, 2) = \mathrm{v6} = 1(1 + f_{\mathrm{io}})$

$p(3, m, 4) = \mathrm{v7} = 1/(1 + f_{\mathrm{oi}})$
$p(3, m, 1) = p(3, no\_m, 1) = \mathrm{v8} = f_{\mathrm{oi}}/(1 + f_{\mathrm{oi}})$
$p(3, no\_m, 7) = \mathrm{v9} = 1/(1 + f_{\mathrm{oi}})$
$p(8, m, 7) = \mathrm{v10} = 1/(1 + f_{\mathrm{oi}})$
$p(8, m, 6) = p(8, no\_nm, 6) = \mathrm{v11} = f_{\mathrm{oi}}/(1 + f_{\mathrm{oi}})$
$p(8, no\_m, 4) = \mathrm{v12} = 1(1 + f_{\mathrm{oi}})$

non-decisional states:

$p(2, n, 4) = \mathrm{v13} = f_{\mathrm{io}}/(1 + f_{\mathrm{io}})$
$p(2, n, 6) = \mathrm{v14} = 1/(1 + f_{\mathrm{io}})$
$p(5, n, 1) = \mathrm{v15} = 1/(1 + f_{\mathrm{io}})$
$p(5, n, 7) = \mathrm{v16} = f_{\mathrm{io}}/(1 + f_{\mathrm{io}})$

$p(4, n, 2) = \mathrm{v17} = f_{\mathrm{oi}}/(1 + f_{\mathrm{oi}})$
$p(4, n, 8) = \mathrm{v18} = 1/(1 + f_{\mathrm{oi}})$
$p(7, n, 3) = \mathrm{v19} = 1/(1 + f_{\mathrm{oi}})$
$p(7, n, 5) = \mathrm{v20} = f_{\mathrm{oi}}/(1 + f_{\mathrm{oi}})$

Table 5
Results of model solution

d	f_{io}	f_{oi}	Optimum solution actions	Optimum solution costs			
	0.01	0.1	1 3 2 3 3 2 3 2	0.089287	0.089285	0.089310	0.089287
				0.089287	0.089285	0.089363	0.089287
	0.001	0.1	1 3 2 3 3 2 3 2	0.009880	0.009881	0.009888	0.009882
				0.009880	0.009881	0.009880	0.009882
	0.001	0.01	1 3 2 3 3 2 3 2	0.090759	0.090744	0.090790	0.090759
				0.090744	0.090744	0.090820	0.090759
	0.1	0.01	2 3 2 3 3 2 3 1	0.089287	0.089363	0.089285	0.089287
				0.089287	0.089310	0.089285	0.089287
100	0.1	0.001	2 3 2 3 3 2 3 1	0.009882	0.009880	0.009881	0.009880
				0.009882	0.009888	0.009881	0.009880
	0.01	0.001	2 3 2 3 3 2 3 1	0.090759	0.090820	0.090744	0.090744
				0.090759	0.090790	0.090744	0.090759
	0.1	0.1	2 3 2 3 3 2 3 2	0.454765	0.454324	0.454324	0.454765
				0.454765	0.454324	0.454324	0.454765
	0.01	0.01	2 3 2 3 3 2 3 2	0.495285	0.494812	0.494812	0.495285
				0.495285	0.494812	0.494812	0.495285
	0.001	**0.001**	**1 3 2 3 3 2 3 1**	**0.100151**	**0.100052**	**0.100149**	**0.100052**
				0.100052	**0.100149**	**0.100052**	**0.100151**
1000	**0.001**	**0.001**	**2 3 2 3 3 2 3 2**	**0.499756**	**0.499268**	**0.499268**	**0.499756**
				0.499756	**0.499268**	**0.499268**	**0.499756**

In Table 5 the results obtained by solving the model under the considered scenarios are reported. Each row corresponds to a different scenario. We distinguish scenarios according to the behavior of the PD mobile device, and the size of the (possibly) mobile MON component. The first column reports the adopted values for the size d of MON, while the second and third columns characterize the PD behavior by means of frequencies of location changes. Finally, the two rightmost columns contain the actual optimal solution: for each state (ordered from 1 to 8) it is shown the action to be performed ($m = 1, no\_m = 2$, $n = 3$) and the minimal average reward per time unit, respectively.[4]

Besides d, that gives the cost of a mobility action, the costs of the other basic system actions for all the considered scenarios are: $p = 1, r = 1$.

The topmost nine rows all refer to a size $d = 100$, and correspond to scenarios that can be subdivided into three categories:

- rows 1 through row 3: *outdoor* scenarios (the PD device is inclined to sojourn more often outdoor than vice versa),
- rows 4 through row 6: *indoor* scenarios (the PD device is inclined to sojourn more often indoor than vice versa),
- rows 7 through row 9: *balanced* scenarios (the willing to move from indoor to outdoor is equal to the one in the opposite direction).

Among the balanced scenarios, only in the one where changes of location are rare (row 9, in bold, in Table 5) mobility is actually indicated somewhere in the solution as an optimal choice. Indeed, we can see that in this scenario the optimal policy chooses the mobility

[4] The minimal reward slightly differs for different states because of the approximation introduced by the adopted iterative solution method [24].

decision in states 1 and 8 that corresponds to moving the MON component from PD to SV when PD is indoor, and to moving MON from SV to PD in the opposite case.

On the contrary, in the other two balanced scenarios (rows 7 and 8) no code mobility is suggested. Evidently in these two scenarios the changes of locations are too frequent to get convenience from migrating, even if the cost of migration is the same.

In all the other scenarios (rows 1 through 6) it can be observed, by generating the optimal Markov chains, that mobility is not a real strategic choice, but it is only indicated in the cases where the position assumed by MON is disadvantageous for the configuration; upon moving in an advantageous position, no chances are left to MON to move once more.

All the above nine experiments have been re-executed changing the costs of the mobility action to $d = 1000$, in order to make mobility an even more costly decision. For all the scenarios except the one corresponding to the ninth row of Table 5, the same optimal solutions (i.e., actions and costs) have been obtained, and therefore they are not reported in the table. For all the indoor and outdoor scenarios, this result validates the concept that mobility only represents a correction for inconvenient transient situations. Where, instead, mobility was suggested as a strategic choice when $d = 100$ (row 9), the increase of its cost leads to change the optimal solution. This is shown in the last row of Table 5 (in bold too), where the result for the same scenario of the row 9 but with $d = 1000$ is included: mobility in this case is excluded from the optimal solution because it is too expensive to be advantegeously used in a systematic way.

5. Conclusions

In this paper we have introduced a new methodology, supported by an appropriate notation, to evaluate the convenience of using a mobility based ADL in a software architecture design. In case of positive answer, the methodology also gives insights into "how" to use mobility to optimize the application performance. Far from coping with all the quality attributes of a mobile code style, our methodology should rather be considered as one among several tools (that should be available to a designer) which are able to address issues and attributes even different from those considered in this paper, Indeed, we are working as well on the definition of methodologies to derive, from a mobility-based SA description, evaluation models addressed to performance indices like response time, not considered here.

Even if our focus was on the performance evaluation methodology, we have also presented some ideas about a suitable formalism to describe such architectures. We have suggested two different notations that can be used to describe the physical and logical mobility, that cope with their different characteristics. Indeed, physical mobility is basically an intrinsic characteristic of some component of the hardware platform, and hence is formally modeled by statements in the "body" of this component. On the other hand, logical mobility is a design option that can affect several quality attributes of the application, including performance. Since in a distributed environment this type of mobility strongly influences the "quality" of the interactions, we have modeled it within a connector, as an optional feature of interactions among components.

We have applied our methodology to a "context-dependent" software application (after having modeled it with the notation here introduced) embedding both physical and logical mobility. The solution of the MDP nicely shows how, in practice, early decisions

about the use of mobility can be effectively taken based on the results of our methodology application.

Like all the performance evaluation methodologies that require the explicit generation of the system state space, also our methodology has to cope with the state explosion problem. The simplification procedure described in Section 4.3 alleviates this problem, but does not eliminate it. Hence this topic deserves further investigation, may be in the direction of only partially generating the system state space, in order to derive an approximate but still significant evaluation of the system performance. Another topic that deserves further investigation is the assignment of probabilities to the MDP state transitions. In the presented methodology, this is a point that requires ingenuity and insight into the system operations to be performed, and hence represents a crucial aspect for the methodology application. We are investigating techniques that allow us to extract at least partial information about the frequencies of different actions from the system specification itself.

Acknowledgements

This work was partially supported by MURST project "SALADIN: Software architectures and languages to co-ordinate distributed mobile components".

Appendix A

Let c be the name of a component instance that appears in the *Components* section; the defined syntax implies that c is a hardware component only if the value of its location variable λ is not a name in the *Components* section, otherwise it is a software component. Hence, the hardware component that hosts a generic component is given by the following function, defined recursively to take into account the general case of nested components

$$hw\_loc(c) = \begin{cases} c & \text{if } c.\lambda \notin \mathit{Components}, \\ hw\_loc(c.\lambda) & \text{if } c.\lambda \in \mathit{Components}. \end{cases}$$

Let $c1$ and $c2$ be the names of two component instances. The type of physical connections existing between them can be given by the following function:

$$conn(c1, c2) = \begin{cases} \mathit{WIRELESS} & \text{if } \mathit{Wireless\_link}(hw\_loc(c1), \\ & \quad hw\_loc(c2)) \in \mathit{Connections}, \\ \mathit{OTHER} & \text{otherwise.} \end{cases}$$

Note that, in our example, it suffices that the $conn(\cdot, \cdot)$ function takes only the two possible values *WIRELESS* and *OTHER*; in general, a richer set of values could be necessary.

Let a be an action executed by the system, corresponding to an arc in the labeled graph LG. In our ADL, a either consists of local assignments to component variables or of the activation of an instance of the *Mob_Comm* connector. We use the notations $a \neq \mathit{MobComm}$ and $a = \mathit{Mob\_Comm}$ (c1, c2, a1, a2, x1, x2, I, M), respectively, to distinguish the two cases; in the latter case, we also use the "dot notation" (e.g., a.c1) to denote the actual values of the connector parameters. Using this notation, the cost of a system action, defined as the number of bytes through a wireless link, is given by the following function:

$$f(a) = \begin{cases} 0 & \text{if } a \neq \mathit{Mob\_Comm}, \\ 0 & \text{if } a = \mathit{Mob\_Comm}(\text{c1, c2, a1, a2, x1, x2, I, } M) \\ & \quad \wedge \mathit{conn}(\text{a.c1, a.c2}) = \mathit{OTHER}, \\ \text{sizeof(a.x1)} & \text{if } a = \mathit{Mob\_Comm}(\text{c1, c2, a1, a2, x1, x2, I, } M) \\ & \quad \wedge \mathit{conn}(\text{a.c1, a.c2}) = \mathit{WIRELESS} \wedge \text{a.M} = \textbf{false}, \\ \text{sizeof(a.c1)} & \text{if } a = \mathit{Mob\_Comm}(\text{c1, c2, a1, a2, x1, x2, I, } M) \\ & \quad \wedge \mathit{conn}(\text{a.c1, a.c2}) = \mathit{WIRELESS} \wedge \text{a.M} = \textbf{true}. \end{cases}$$

In our example we have used the following notations for the values taken by the sizeof function:

sizeof(req) = r
sizeof(pos) = p
sizeof(MON) = d.

Appendix B

The technique we present in this appendix for the simplification of an MDP can be considered as a generalization to the case of MDP of analogous techniques for the simplification of Markov processes or Markov reward processes (see for example [15] for a techniques for the elimination of "vanishing states" from the Markov process obtained from a Generalized Stochastic Petri Net).

Let us consider a particular policy $\sigma : S \to A$ in the original MDP, and let us define a policy $\hat{\sigma} : \hat{S} \to \hat{A}$ for the simplified MDP as follows:

$$\hat{\sigma}(s) = \sigma(s), \quad s \in \hat{S}. \tag{B.1}$$

Let $\mathbf{P}(\sigma) = [p(s, \sigma(s), s')]$ be the $|S| \times |S|$ transition probability matrix of the Markov process obtained from the MDP when policy σ is adopted, and let $\pi(\sigma) = [\pi(s, \sigma)]$ be the row vector of its steady-state probabilities. If the states of S are re-ordered according to the subsets S_N and S_0, we can write

$$\mathbf{P}(\sigma) = \begin{bmatrix} \mathbf{P}_{NN}(\sigma) & \mathbf{P}_{N0}(\sigma) \\ \mathbf{P}_{0N}(\sigma) & \mathbf{P}_{00}(\sigma) \end{bmatrix}, \quad \pi(\sigma) = \big[\pi_N(\sigma), \pi_0(\sigma)\big].$$

Note that the matrices $\mathbf{P}_{00}(\sigma)$ and $\mathbf{P}_{0N}(\sigma)$ are actually independent of σ, since by definition the subset S_0 is made of non-decisional states, where only decision n is enabled. Hence, we can write $\mathbf{P}_{00}(\sigma) = \mathbf{P}_{00}$ and $\mathbf{P}_{0N}(\sigma) = \mathbf{P}_{0N}$. Moreover, since all the outgoing transitions from states of S_0 have reward equal to zero, the steady-state average reward per unit time is given by

$$g(\sigma) = \sum_{s \in S_N} \pi(s, \sigma) r(s, \sigma(s)) \tag{B.2}$$

Vector $\pi(\sigma)$ satisfies the following balance equation, up to a multiplicative constant [24]

$$\pi(\sigma) = \pi(\sigma)\mathbf{P}(\sigma)$$

which, with the above decomposition, can be rewritten as

$$[\pi_N(\sigma), \pi_0(\sigma)] = [\pi_N(\sigma), \pi_0(\sigma)] \begin{bmatrix} \mathbf{P}_{NN}(\sigma) & \mathbf{P}_{N0}(\sigma) \\ \mathbf{P}_{0N} & \mathbf{P}_{00} \end{bmatrix}$$

From this last equation we get

$$\begin{cases}\pi_{\mathrm{N}}(\sigma) = \pi_{\mathrm{N}}(\sigma)\mathbf{P}_{\mathrm{NN}}(\sigma) + \pi_0(\sigma)\mathbf{P}_{0\mathrm{N}} \\ \pi_0(\sigma) = \pi_{\mathrm{N}}(\sigma)\mathbf{P}_{\mathrm{N}0}(\sigma) + \pi_0(\sigma)\mathbf{P}_{00}\end{cases}$$

From the second vector–matrix equation in the system above, we have

$$\pi_0(\sigma) = \pi_{\mathrm{N}}(\sigma)\mathbf{P}_{\mathrm{N}0}(\sigma)(\mathbf{I} - \mathbf{P}_{00})^{-1}$$

and, substituting this last equation into the first equation of the system above, we get

$$\pi_{\mathrm{N}}(\sigma) = \pi_{\mathrm{N}}(\sigma)(\mathbf{P}_{\mathrm{NN}}(\sigma) + \mathbf{P}_{\mathrm{N}0}(\sigma)(\mathbf{I} - \mathbf{P}_{00})^{-1}\mathbf{P}_{0\mathrm{N}}) \tag{B.3}$$

Now, let us consider the Markov process obtained from the simplified MDP defined in Section 4.3, with the policy $\hat{\sigma}$ defined by (4), and let $\hat{\mathbf{P}}(\hat{\sigma})$ and $\hat{\pi}(\hat{\sigma}) = [\hat{\pi}(s, \sigma)]$ be the corresponding transition probability matrix and steady-state probability row vector, respectively. Vector $\hat{\pi}(\hat{\sigma})$ satisfies the following balance equation, up to a multiplicative constant [24]

$$\hat{\pi}(\hat{\sigma}) = \hat{\pi}(\hat{\sigma})\hat{\mathbf{P}}(\hat{\sigma}). \tag{B.4}$$

The steady-state average reward per unit time of this process is given by

$$\hat{g}(\hat{\sigma}) = \sum_{s \in \hat{S}} \hat{\pi}(s, \hat{\sigma})\hat{r}(s, \hat{\sigma}(s)). \tag{B.5}$$

From the definition of the transition probability function : $\hat{P} : \hat{S} \times \hat{A} \times \hat{S} \to [0, 1]$ given in Section 4.3, it is easy to realize that

$$\hat{\mathbf{P}}(\hat{\sigma}) = \mathbf{P}_{\mathrm{NN}}(\sigma) + \mathbf{P}_{\mathrm{N}0}(\sigma)(\mathbf{I} - \mathbf{P}_{00})^{-1}\mathbf{P}_{0\mathrm{N}}. \tag{B.6}$$

Hence, it follows from (B.3), (B.4) and (B.6) that $\hat{\pi}(\hat{\sigma})$ and $\pi_{\mathrm{N}}(\sigma)$ differ only by a multiplicative constant.

Let us consider any two policies $\sigma' : S \to A$ and $\sigma'' : S \to A$, and the corresponding policies $\hat{\sigma}' : \hat{S} \to \hat{A}$ and $\hat{\sigma}'' : \hat{S} \to \hat{A}$ defined according to (1). From the definition of Section 4.3 we know that $\hat{r}(s, a) = r(s, a)$, $s \in \hat{S}$, $a \in A(s)$. Moreover, we have shown above that $\hat{\pi}(\hat{\sigma})$ and $\pi_{\mathrm{N}}(\sigma)$ differ only by a multiplicative constant. As a consequence, we have that $g(\sigma') \leqslant g(\sigma'')$ if and only if $\hat{g}(\hat{\sigma}') \leqslant \hat{g}(\hat{\sigma}'')$, with $g(\cdot)$ and $\hat{g}(\cdot)$ defined by (B.2) and (B.5), respectively.

This implies that the optimal policy for both the original MDP and the simplified one is the same.

References

[1] F. Andolfi, F. Aquilani, S. Balsamo, P. Inverardi, Deriving performance models of software architectures from message sequence charts, in: Proceedings of the 2nd Workshop on Software and Performance (WOSP2000), Ottawa, Canada, September 17–20, 2000.

[2] M. Baldi, G.P. Picco, Evaluating the tradeoffs of mobile code design paradigms in network management applications, in: R. Kemmerer, K. Futatsugi (Eds.), Proceedings of the 20th International Conference on Software Engineering (ICSE 98), Kyoto, Japan, 1998.

[3] S. Balsamo, P. Inverardi, C. Mangano, An approach to performance evaluation of software architectures, in: Proceedings of the Workshop on Software and Performance (WOSP '98), Santa Fe, New Mexico, October 12–16, 1998.

[4] L. Bass, P. Clements, R. Kazman, Software Architectures in Practice, Addison-Wesley, New York, NY, 1998.

[5] P.J. Brown, J.D. Bovey, X. Chen, Context-aware application from the laboratory to the marketplace, IEEE Personal Commun. 4 (5) (1997) 58–64.

[6] L. Cardelli, A.D. Gordon, Mobile ambients, in: M. Nivat (Ed.), Foundations of Software Science and Computational Structures, Lecture Notes in Computer Science, vol. 1378, Springer, Berlin, 1998, pp. 140–155.
[7] M.-H. Chen, M.-H. Tang, W.-L. Wang, Effect of architecture configuration on software reliability and performance estimation, in: Proceedings of the IEEE Workshop on Application-specific Software Engineering and Technology (ASSET 98), Richardson, TX, March 1998.
[8] V. Cortellessa, R. Mirandola, Deriving a queueing network based performance model from UML diagrams, in: Proceedings of the 2nd Workshop on Software and Performance (WOSP2000), Ottawa, Canada, September 17–20, 2000.
[9] R. De Nicola, G. Ferarri, R. Pugliese, B. Venneri, KLAIM: a kernel language for agents interaction and mobility, IEEE Trans. Software Engrg. 24 (5) (1998) 315–330.
[10] C. Drane, C. Rizos, Positioning systems in intelligent transportation systems, Intelligent Transportation Systems, Artech House, 1998.
[11] A. Fuggetta, G.P. Picco, G. Vigna, Understanding code mobility, IEEE Trans. Software Engrg. 24 (5) (1998) 342–361.
[12] H. Grahn, J. Bosch, Some initial performance characteristics of three architectural styles, in: Proceedings of the Workshop on Software and Performance (WOSP '98), Santa Fe, New Mexico, October 12–16, 1998.
[13] D. Kotz, G. Jiang, R. Gray, G. Cybenko, R.A. Peterson, Performance analysis of mobile agents for filtering data streams on wireless networks, in: Proceedings of the 3rd International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2000), August 2000.
[14] R. Kravets, P. Krishnan, Power management techniques for mobile communications, in: Proceedings of the 4th ACM/IEEE Conference on Mobile Computing and Networking (MOBICOM 98), Dallas, TX, October 1998.
[15] M.A. Marsan, G. Conte, A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems, ACM Trans. Comput. Systems 2 (2) (1984) 93–122.
[16] N. Medvidovic, R.N. Taylor, A classification and comparison framework for software architecture description languages, IEEE Trans. Software Engrg. 26 (1) (2000) 70–93.
[17] A. Puliafito, S. Riccobene, M. Scarpa, An analytical comparison of the client–server, remote evaluation and mobile agent paradigms, in: Proceedings of the Ist International Symposium on Agent Systems and Applications and 3rd International Symposium on Mobile Agents (ASA/MA 99), October 1999.
[18] G.-C. Roman, P.J. McCann, An introduction to mobile UNITY, in: J. Rolim (Ed.), Parallel and Distributed Processing, Lecture Notes in Computer Science, vol. 1388, Springer, Berlin, 1998, pp. 871–880.
[19] G. Samaras, M.D. Dikaiakos, C. Spyrou, A. Liverdos, Mobile agent platforms for Web databases: a qualitative and quantitative assessment, in: Proceedings of the Ist International Symposium on Agent Systems and Applications and 3rd International Symposium on Mobile Agents (ASA/MA 99), October 1999.
[20] T. Spalink, J.H. Hartman, G. Gibson, The effects of a mobile agent on file service, in: Proceedings of the Ist International Symposium on Agent Systems and Applications and 3rd International Symposium on Mobile Agents (ASA/MA 99), October 1999.
[21] B. Spitznagel, D. Garlan, Architecture-based performance analysis, in: Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering, June 1998.
[22] M. Stemm, R.H. Katz, Measuring and reducing energy consumption of network interfaces in hand-held devices, IEICE Trans. Commun. (1997) (special issue on Mobile Computing).
[23] M. Strasser, M. Schwehm, A performance model for mobile agent system, in: H.R. Arabnia (Ed.), International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 97), Las Vegas, vol. II, 1997, pp. 1132–1140.
[24] H.C. Tijms, Stochastic Modelling and Analysis: A Computational Approach, Wiley, New York, 1986.
[25] R. Want, A. Hopper, Active badges and personal interactive computing objects, IEEE Trans. Consumer Electron. 38 (1) (1992).
[26] M. Wermelinger, J.L. Fiadeiro, Connectors for mobile programs, IEEE Trans. Software Engrg. 24 (5) (1998) 331–341.
[27] L.G. Williams, C.U. Smith, Performance evaluation of software architectures, in: Proceedings of the Workshop on Software and Performance (WOSP '98), Santa Fe, New Mexico, October 12–16, 1998.